Calculator ISA Report

Abraham McIlvaine

Abstract: The goal of this project was to design an 8-bit ISA for a calculator, and to create a simulator in C to be able to run programs written using our ISA design. Functions of the ISA included loading immediate values into registers, adding and subtracting registers, printing register contents to the console, and comparing two registers and skipping either 0,1, or 2 instructions based on the comparison result. The final result of this project is a fully functioning simulator for our ISA that we designed.

Division of Labor: We divided the labor exactly as was specified by the project description sheet. The design of the ISA was done with a partner, then we each individually created our own simulator for the same ISA. Then we tested our partner's simulator using benchmarks that we created. Lastly, we each individually created a report to describe the process and results of the project.

Detailed Strategy:

ISA Design

When we were designing the ISA, we were constrained to 8 bits for each instruction length and this presented some problems. For adding and subtracting, we knew that 6 bits would be needed to address 3 different registers. This left 2 bits for the op-code. For loading an immediate 4 bit value, 2 bits would also be left for the op-code. However, we needed to distinguish between 5 different instructions. Our solution to this problem was to use a variable length op-code. The op-codes for the 5 different ISA functions are shown in the table below.

Function	Op-Code
add	01
sub	10
load imm.	11
compare	001
print	000

By defining the op-codes in this way, we were able to distinguish all functions from one another while also allowing necessary bits for required fields for the functions.

Simulator Implementation

The simulator was designed using C, as was recommended by the project description. It was straightforward to implement; there were no problems that caused me too much trouble. The basic design of the simulator is to first check which function needs to be executed by checking the op-code. Then, the corresponding function is called to simulate the function encoded in the text file.

```
//runs each instruction
while(next_i()){
    switch (func()){
        case ADD:
            add();
            break;
        case SUB:
            sub();
            break;
        case LI:
            li();
            break;
        case CMP:
            cmp();
            break;
        case PRINT:
            print();
            break;
    }
}
```

The code above is the heart of the simulator. The function next_i() reads the next instruction until there are no more instructions to execute. The function func() checks the op-code, which then allows the corresponding function to be called to handle the correct ISA function.

Below is the code for func(), the function which decides the correct function to call. The array “i” is where the current instruction data is stored, and the indexes 0,1, and 2 are where the op-codes can be stored. The function checks for the correct op-code and returns an integer representing the correct function. If the op-code is invalid, the function bad_i() is called, which terminates the simulator.

```
int func(){
    if(i[0]=='0' && i[1]=='1'){
        return ADD;
    }
    else if(i[0]=='1' && i[1]=='0'){
        return SUB;
    }
    else if(i[0]=='1' && i[1]=='1'){
        return LI;
    }else if(i[0]=='0' && i[1]=='0'){
        if(i[2]=='1'){
            return CMP;
        }
        else if(i[2]=='0'){
            return PRINT;
        }
    }
    else{
        bad_i();
    }
}
```

One problem that was encountered when creating the simulator was related to how I was reading in the file. I did not account for the new-line character at the end of each line, and when reading in a line the data ended up spilling over into the registers data, corrupting the register data for the simulator. This problem was fixed by simply adding one more spot to the array of characters each instruction was read in to.

Benchmark Design

Benchmarks were designed to isolate each function and test them without dependence on other functions, although the print function is needed everywhere to check the value of registers. Each benchmark tested all cases and each instruction's major variations. Major variations of instructions tested include skipping 0,1,or 2 lines for compare and the correctness of sign extension when loading a 4-bit immediate value.

When running my benchmarks on my partner's simulator, I only found one problem. This problem was that when his print function printed out a register value in hexadecimal, more bits were printing than are present in the 8-bit registers. This was fixed by adding a "& 0xFF" statement. My partner is also my roommate, so I communicated with him directly.

My partner did not find any problems with my simulator using his benchmarks.

Results:

Running the benchmarks on the simulator, I found that no problems occurred. This was to be expected because the simulator was not complicated. The simulator functioned exactly as planned, and there were no errors in the ISA design.

Conclusion:

This project was very beneficial to our understanding of the process of designing an ISA for something such as an 8-bit calculator. Although this was a simple project, the ideas and design strategies extend to much larger design problems, such as a CPU for a computer. The implementation of the simulator was a good exercise in C programming. We learned about file handling, code organization, and other concepts. Overall, this project increased our understanding of computer system design.

Appendix I (notebooks):

Team member	Hours spent
Abraham	6
Ben	9

Appendix II (source files):

See files included in submission folder. Included is the source code and the benchmarks.

HOW TO USE THE SIMULATOR:

If the file is compiled as "simulator", then it is run by:

```
 ./simulator {name of file}
```

"name of file" is the file with the list of commands, one per line.

for example a commands file could look like this:

```
11000001
11010010
01100001
00010100
01101000
00010100
11111000
11000111
01011100
00001100
```

We chose to use a textual representation of the instructions.